Scalable Distributed Consensus to Support MPI Fault Tolerance

Darius Buntinas
Argonne National Laboratory
buntinas@mcs.anl.gov

***Abstract*—As system sizes increase, the amount of time in which an application can run without experiencing a failure decreases. Exascale applications will need to address fault tolerance. In order to support algorithm-based fault tolerance, communication libraries will need to provide fault-tolerance features to the application. One important fault-tolerance operation is distributed consensus. This is used, for example, to collectively decide on a set of failed processes. This paper describes a scalable, distributed consensus algorithm that is used to support new MPI fault-tolerance features proposed by the MPI 3 Forum's fault-tolerance working group. The algorithm was implemented and evaluated on a 4,096-core Blue Gene/P. The implementation was able to perform a full-scale distributed consensus in 222 μs and scaled logarithmically.**

I. Introduction

As process counts in applications grow toward exascale, the length of time an application can run without experiencing a failure, known as the mean time between failures (MTBF), decreases. Applications will need to address fault tolerance in order to be useful on future exascale machines. Checkpointing can provide fault tolerance to an application without the need to modify it. However, as the MTBF decreases, checkpoints will need to be taken more often, thus decreasing the amount of useful work the application can perform between failures.

Whereas checkpointing provides fault tolerance to an application in a transparent manner, when using algorithm-based fault tolerance (ABFT) [1][2][3], the application is aware of faults and handles them explicitly. The fault-tolerance working group of the MPI 3 Forum has been working on a proposal [4], that adds fault-tolerance features to MPI in order to support ABFT applications. The proposal defines the behavior of an MPI library if processes fail. For example, existing operations such as MPI_Comm_split are required by the proposal to either succeed at every process or return an error at every process, even if processes fail before or during the operation. The proposal also introduces new functions, such as MPI_Comm_validate, that require all processes to return the same list of failed processes. A distributed consensus algorithm is needed to implement these operations.

This paper presents a scalable, fault tolerant, distributed consensus algorithm used to implement the MPI_Comm_validate function. The MPI_Comm_validate implementation is evaluated on a 4,096-core IBM Blue Gene/P machine and exhibits $\mathcal{O}(\log n)$ scaling.

This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

The rest of the paper proceeds as follows. In Section II we describe the problem. In Section III the algorithm is presented along with proofs. In Section IV we describe the implementation of MPI_Comm_validate using the distributed consensus algorithm. In Section V we evaluate the performance. In Section VI we review related work. In Section VII we conclude the paper and briefly discuss future work.

II. Problem Description

We present the distributed consensus algorithm as it would be used in the MPI_Comm_validate operation, although the algorithm could also be used in other operations requiring distributed consensus, such as MPI_Comm_split. We first list the assumptions we make on the environment and then describe the MPI_Comm_validate function.

Assumptions on the environment:

1) The only failures will be process failures. Communication errors are masked by the MPI implementation. We do not consider network partitioning in this paper.
2) Process failures will be fail-stop failures. Once a process fails, it will stop sending messages.
3) Failure detectors are *eventually perfect* [5] with the additional requirement that if any process suspects a process to have failed, then it will be suspected permanently and will eventually be suspected by all processes.
4) Processes do not spontaneously recover after failure. Once a process has failed, it will remain failed.
5) There will always be a point in time in the future when no processes fail long enough to allow the algorithm described to complete.

The MPI_Comm_validate function uses distributed consensus to decide on a set of failed processes, which must contain every failed process known by any participating process at the time the function is called. The same set of failed processes must be returned by the function at every process. If a process fails during the MPI_Comm_validate operation (i.e., after the first process calls the function and before any process returns), the set of failed processes returned may or may not contain that failed process.

A. Failure Detector

The MPI 3 fault-tolerance proposal requires an *eventually perfect* failure detector [5]. An eventually perfect failure detec-

tor guarantees that if a process fails, every other process will eventually suspect that process of having failed, and that if a process is alive, then eventually no process will suspect it. This means that it is possible for a live process to be mistakenly suspected of having failed, but that eventually every process will realize that it is still alive. Since the failure detector may mistakenly report a process as failed, a process can never be sure that the process has in fact failed, so in this paper we say that the process is *suspected* of having failed, or that the process is *suspect*.

The proposal adds an additional requirement to handle the case of processes that are mistakenly believed to have failed. The requirement says that if a process is incorrectly suspected as having failed, then every process will eventually suspect that process. In addition, the MPI implementation is allowed to kill any processes that are mistakenly identified as failed. This requirement means that algorithms do not have to handle the case where a process that was once suspected is no longer suspected. Although this requirement seems like a heavy handed way of dealing with false positives, we expect such cases will not be very common. Exascale systems will have RAS systems in place to monitor various components, and can more reliably detect hardware failures than by relying on timeouts. The proposal also requires that when a process suspects that another process has failed, it will no longer receive messages from the suspected process, even if the process is still alive.

This paper does not consider transient failures or network failures, including network partitioning. Such failures are not considered by the current MPI 3 fault-tolerance proposal, but the fault-tolerance working group is planning on addressing them in future proposals. This paper does not address the implementation of a failure detector, but does assume the presence of a failure detector as described above.

B. Loose Semantics

The MPI 3 fault-tolerance proposal discusses allowing *loose semantics* for the MPI_Comm_validate operation. Consider the case where a process returns a set of failed processes, and then immediately fails before the consensus protocol completes. Loose semantics allow the remaining live processes to agree on a set of failed processes which is different from that returned by the failed process. We expect that many applications will not require the strict semantics, since the remaining live processes all returned the same set, and by using loose semantics the latency of the algorithm is improved by eliminating a phase.

III. Algorithm

In this section, we give a brief overview of the algorithm at a high level and then describe the broadcast and distributed consensus algorithms in detail.

The algorithm proceeds in three phases. In Phase 1, the root generates a ballot and broadcasts it to the other processes. Each process then responds with an `ACCEPT` or `REJECT` depending on whether it finds the proposed ballot acceptable. The response is collected up to the root. If the ballot is rejected, the root generates a new ballot and tries again. Once all processes accept a ballot, the root proceeds to Phase 2 where it broadcasts an `AGREE` message to all processes. Once a process receives an `AGREE` message, it knows that the ballot has been agreed upon by all processes. In Phase 3, the root broadcasts a `COMMIT` message. Upon receiving the `COMMIT` message, the processes commit to the ballot, and can return from the MPI_Comm_validate.

A. Basic Fault Tolerant Tree Broadcast Algorithm

The fault tolerant tree broadcast algorithm is used to ensure messages are received by all processes in the presence of process failure. The algorithm returns either an `ACK` or a `NAK` to the root process indicating whether the algorithm succeeded. If the algorithm returns an `ACK`, then all processes have received the message. Listing 1 shows the algorithm.

The algorithm is initiated by the root, which is the lowest ranked unsuspected process. The root chooses a value for `bcast_num` that is larger than any `bcast_num` value that it has used or seen previously. The `bcast_num` value is included in all messages to ensure that messages from a previously aborted instance of the broadcast algorithm do not interfere with the current instance. The root then computes its set of descendant processes (line 4) and calls the compute_children function in line 16 to compute the set of children as well as assigning descendant processes to each child. The compute_children function is described below. The root then sends a `BCAST` message to each child that contains the `bcast_num` along with the set of descendants for that child (line 18).

Non-root processes wait for a `BCAST` message (line 7). If an old `BCAST` message is received, the process replies with a `NAK` message to the sender. A `NAK` is sent, rather than just ignoring the message so that if the root process did not choose a `bcast_num` that was large enough and it was used previously, then the root will not hang but will receive a `NAK` and can try again. Once the child receives a `BCAST` message with an acceptable `bcast_num` it sets its own `bcast_num` to that value, and sets its parent set of descendants. The process, then computes its set of children and sends `BCAST` messages to its children (line 16).

The root and non-root processes then wait for acknowledgments from each child (line 22). If a child should fail while the process is waiting for an acknowledgment, a `NAK` is sent to the parent (lines 24).

If for some reason, a new instance of the broadcast algorithm is initiated, it is possible that a process may receive a `BCAST` message while waiting for an acknowledgment from its children (line 26). Non-root processes respond by sending a `NAK` if the `bcast_num` of the message is not larger than the process's `bcast_num`, but it the messages `bcast_num` is larger, then the process abandons the current instance and starts the algorithm over (line 31).

If a process receives a `NAK` from a child, a `NAK` is sent to the parent (lines 35). If all children return an `ACK`, then the

Listing 1. Fault tolerant broadcast algorithm

```
1  if (rank = root)
2    parent ← NULL
3    bcast_num ← larger than any bcast_num seen
4    my_descendants ← {p ∈ processes : rank(root) < rank(p) < num_procs}
5  else
6    do
7      wait for BCAST message
8      if (msg.bcast_num ≤ bcast_num)
9        send NAK to msg.sender
10   while (msg.bcast_num ≤ bcast_num)
11 L1:
12   bcast_num ← msg.bcast_num
13   my_descendants ← msg.descendants
14   parent ← msg.sender
15
16 compute_children(my_descendants)
17 for each child ∈ children
18   send BCAST with bcast_num and descendants[child] to child
19
20 num_ACKs ← 0
21 while (num_ACKs < |children|)
22   wait for ACK/NAK message or child failure
23   if (child fails)
24     send NAK with bcast_num to parent
25     return NAK
26   if (BCAST received)
27      if (msg.bcast_num ≤ bcast_num) /* NAK old bcasts */
28        send NAK with msg.bcast_num to sender
29        continue
30      else /* new bcast has been initiated */
31        goto L1
32   if (msg.bcast_num ≠ bcast_num)
33     continue
34   if (NAK received)
35     send NAK to parent
36     return NAK
37   ++num_ACKs
38
39 send ACK with bcast_num to parent
40 return ACK
```

Listing 2. Algorithm for computing children

```
1  compute_children(my_descendants)
2    while (my_descendants ≠ ∅)
3      do
4        choose child ∈ my_descendants
5        my_descendants ← my_descendants − {child}
6      while (child is suspect)
7      descendants[child] ← {p ∈ my_descendants : rank(p) > rank(child)}
8      my_descendants ← my_descendants − descendants[child]
9      children ← children ∪ {child}
```

process sends an ACK to its parent (line 39) and the algorithm returns with an ACK.

Because of the way descendant sets are computed, a parent's rank will always be lower than any of its children's ranks. This means that if a process suspects all processes with lower ranks and appoints itself root, it cannot receive a BCAST message, even if it incorrectly suspected a process with a lower rank, so root processes will never need to handle received BCAST messages.

The set of children is computed by using the compute_children function shown in Listing 2. Given a set of descendants, the process chooses a child from that set and assigns all processes from the descendant set with ranks higher than the child's to the child's descendant set. The child and its descendants are removed from the process's descendant set. The process repeats until the process's descendant set is empty. Any suspected children are discarded.

Note that when choosing a child from its descendant set, if a process always chooses a descendant with a rank closest to the median rank, this broadcast algorithm will generate a binomial tree.

The broadcast algorithm satisfies three properties:

1) (Correctness) If the algorithm returns an ACK at the root process, then all non-suspect processes received the BCAST message.
2) (Termination) The root process of the instance of the algorithm with the largest bcast_num will return either an ACK or a NAK.
3) (Non-triviality) If no processes become suspect during the execution of the algorithm, then the instance of the algorithm with the largest bcast_num will return an ACK at the root.

We first prove several lemmas then prove that the broadcast algorithm satisfies these properties.

Lemma 1: If no processes become suspect while the broadcast algorithm is running, the instance of the algorithm with the highest bcast_num constructs a spanning tree reaching every live process.

Proof: By induction.

Base case (no descendants): A process with an empty descendant set (i.e., a process with no children) is a spanning tree.

Inductive step ($n+1$ descendants): In compute_children, process's descendant set is divided into sets containing a child and its descendants. The process will send a BCAST message to each child, and upon receiving the message, each child will set its parent to that process. Since each child forms a spanning tree with its descendants, the process and its descendants also forms a spanning tree. Because this instance of the broadcast algorithm has the highest bcast_num, no other message can change the parent of any child, and the tree will remain a spanning tree until another broadcast with a higher bcast_num is performed.

Therefore the algorithm constructs a spanning tree of a process and its descendants. ■

Lemma 2: Consider the instance of the broadcast algorithm with the largest bcast_num. If while the algorithm is running, a process that has not yet been added to the spanning tree becomes suspected, then the algorithm will still create a spanning tree of the remaining non-suspect processes.

Proof: The algorithm is unaffected by a process becoming suspect before it is added to the spanning tree because the algorithm always chooses children from the non-suspect processes in the descendant set. ■

Lemma 3: If a process that has already been added to the spanning tree is suspected by its parent before sending an ACK, the root will not receive an ACK from every child, but will receive a NAK from some child or suspect a child.

Proof: If process A is suspected by its parent, the parent of process A will send a NAK rather than an ACK to its parent. The NAK will be forwarded up the tree to the root. If the NAK cannot be forwarded to the root because some process, B, between process A and the root fails, then the parent of process B will suspect its child and send a NAK to its parent which will be forwarded up the tree. Since process B will always be closer to the root than process A, even if there are multiple failures along the route, a NAK will either eventually reach the root or the root will suspect one of its own children. In either case, since a process will not send an ACK after sending a NAK, the root will not receive an ACK from every child. ■

Lemma 4: In a spanning tree, if a process sends an ACK, then it and all of its descendants have received the message.

Proof: By induction.

Base case (leaf process): A leaf process will only send an ACK when it has received the BCAST message. It has no descendants, so it and all of its descendants have received the message.

Inductive step: A non-root process of the spanning tree will send an ACK only when it has received an ACK from each child. By the induction hypothesis, if the process received an ACK from every child, then every descendant of the process has received the message. The process itself must have received the BCAST message because it had forwarded the message to its children. ■

We now prove the correctness, termination and non-triviality properties of the broadcast algorithm.

Theorem 1: (Correctness) If the root returns with an ACK, then all non-suspect processes have received the BCAST message.

Proof: From Lemma 4, if the root receives an ACK from every child, then every child and every child's descendants have received the BCAST message. The compute_children function assigns every process in the root's descendant set to the set of children or to the descendants of a child, so every descendant of the root has received the BCAST message. The root's descendant set consists of all processes with rank lower than the root (line 4), and the root would not have appointed itself root unless it suspected every process with a lower rank. The only way a process returns from the algorithm with an ACK is if it has received an ACK from every child (line 40). Therefore, if the root receives an ACK from every child, then

all non-suspect processes have received the BCAST message, and the root returns with an ACK. ■

Theorem 2: (Termination) The non-suspected root will return either an ACK or a NAK.

Proof: If no process fails after being added to the tree, then by Lemmas 2 and 4, a spanning tree will be created and the root will receive an ACK from every child and will return an ACK. If a process fails after being added to the tree but before sending an ACK, then by Lemma 3, the root will receive a NAK from a child or, if a child fails, the root will suspect the child. In either case the root will return a NAK (lines 25 and 36).

The termination of the algorithm will not be affected in the case where a process becomes suspect after sending an ACK, because the process has already sent an ACK and it's parent process will not be waiting for its acknowledgment.

In the case where the root receives an ACK from every child, it will return an ACK. In the case where the root receives a NAK from some child, or if it suspects a child, then it will return a NAK. ■

Theorem 3: (Non-triviality) If no processes are suspected during execution, then all processes will receive the BCAST message from the instance of the algorithm with the largest bcast_num.

Proof: By Lemma 1 if no process is suspected, during the execution of the instance of the algorithm with the largest bcast_num, then a spanning tree will be created.

Every process in the tree will receive the message:

Base case (root node, depth 0): The root process will send the message to every child.

Inductive step (depth $n+1$): When a process receives the message, it will forward it to each child.

Since the tree is a spanning tree, every process will receive the message.

■

B. Distributed Consensus Algorithm

The distributed consensus algorithm is shown in Listing 3. The listing is split into three parts: introductory steps performed by all processes, the steps performed by the root process and the actions performed by non-root processes. The steps performed by the root are executed serially starting at Phase 1. The non-root part of the listing shows four actions; each action consists of an event followed by statements. The statements of an action are executed by a process when the associated event occurs, e.g., when a message is received.

The distributed consensus algorithm uses a modified version of the broadcast algorithm to distribute ballots and collect responses. The algorithm used for broadcasting ballots is similar to the regular broadcast algorithm except that (1) a ballot is piggybacked on the BCAST messages, (2) a response is piggybacked on the ACK messages (3) when a process receives an ACK from every child, if every ACK message had an ACCEPT response piggybacked on it and the process itself found the ballot acceptable, then the process piggybacks an ACCEPT on the ACK message that it sends to its parent;

Listing 3. Distributed consensus algorithm

```
1  Initialization:
2      state ← BALLOTING
3      root is lowest ranked non−suspect proc
4  Root Process (start in Phase 1)
5  Phase 1:
6      generate ballot
7      r ← broadcast BCAST(BALLOT) with ballot
8      if (r = NAK(AGREE_FORCED))
9        ballot ← msg.ballot
10       goto Phase 2
11     if (r = NAK)
12       restart Phase 1
13     if (r = ACK(REJECT))
14       restart Phase 1
15     goto Phase 2
16
17 Phase 2:
18     state ← AGREED
19     r ← broadcast BCAST(AGREE) with ballot
20     if (r = NAK)
21       restart Phase 2
22     goto Phase 3
23
24 Phase 3:
25     state ← COMMITTED
26     r ← broadcast BCAST(COMMIT)
27     if (r = NAK)
28       restart Phase 3
29
30 Non-Root Processes (select any enabled action)
31 Recv BCAST(BALLOT):
32     if (state = BALLOTING)
33       broadcast algorithm
34     else
35       send NAK(AGREE_FORCED) with ballot
36
37 Recv BCAST(AGREE):
38     if (state ≠ BALLOTING and
39         ballot ≠ msg.ballot)
40       send NAK to sender
41     broadcast algorithm
42     ballot ← msg.ballot
43     state ← AGREED
44
45 Recv BCAST(COMMIT):
46     broadcast algorithm
47     state ← COMMITTED
48
49 Suspect all processes with rank less than
   self:
50     appoint self as root process
51     if (state = COMMITTED)
52       goto Phase 3
53     else if (state = AGREED)
54       goto Phase 2
55     else
56       goto Phase 1
```

otherwise the process piggybacks a REJECT on the ACK, and (4) if a process receives an AGREE_FORCED message piggybacked on a NAK message, it will also piggyback the AGREE_FORCED message on the NAK message it sends to its parent. In Listing 3, when a message has a piggybacked message, the piggybacked message will be written in parentheses after it, e.g., ACK(REJECT) is a REJECT piggybacked on an ACK.

The algorithm begins by initializing state to BALLOTING at every process, and by appointing the lowest ranked non-suspect process as root. In Phase 1 the root generates a ballot and broadcasts the ballot (line 7). When a non-root process receives the ballot (line 31), if it has not already agreed to another ballot, it performs the ballot broadcast algorithm as described above. If the process has already agreed to another ballot (line 35), then it sends a NAK message with a piggybacked AGREE_FORCED message along with the previously agreed upon ballot. The NAK(AGREE_FORCED) message is forwarded up the tree to the root.

If the root receives a NAK(AGREE_FORCED) message it records the ballot and jumps to Phase 2. If the broadcast returned a NAK because of a failed process or if the ballot was rejected, the root restarts Phase 1 with a new ballot. Otherwise, the ballot was accepted and the root jumps to Phase 2. (line 15).

In Phase 2, the root knows that the ballot has been agreed upon by all processes. The root sets its state to AGREED, then broadcasts an AGREE message along with the ballot (line 19). Upon receiving the AGREE message (line 37), a process executes the broadcast algorithm, saves the ballot and sets its state to AGREED. If the broadcast algorithm returns a NAK at the root, indicating that some process has failed, the root restarts Phase 2, otherwise the root jumps to Phase 3.

When a process is in the AGREED state, it knows that all processes have agreed on the ballot. When the root starts Phase 3, it knows that every process is in the AGREED state. The root starts Phase 3 by setting its state to COMMITTED, then broadcasting the COMMIT message (line 26). When a process receives the COMMIT message (line 45), it executes the broadcast algorithm, then sets its state to COMMITTED. The root will restart Phase 3 until the broadcast succeeds with an ACK.

The algorithm handles non-root process failure by repeating broadcast operations. However, when a root process fails, a new root must appoint itself. This is done in line 49 when a process detects that all lower ranked processes are suspect. Depending on its state, the new root will start the algorithm at one of the three phases. If the new root is in the COMMITTED state, then it knows that all processes are in either the COMMITTED or AGREED states, so it can jump to Phase 3. If the new root is in the AGREED state, then it knows that all processes have agreed to the ballot, so it can jump to Phase 2. If the new root is in the BALLOTING state, then it does not know if a ballot has been agreed upon but it does know that no process can be in the COMMITTED state, so it jumps to Phase 1. If it turns out that some process was in the AGREED state, because it received an AGREE message before the old root became suspect, then that process will reply with a NAK(AGREE_FORCED) message when it receives the ballot (line 35). When the root receives the NAK(AGREE_FORCED) message, it knows that a previous ballot had been agreed upon so it will jump to Phase 2 (line 10). Note that process failures might prevent the NAK(AGREE_FORCED) message from reaching the root, however as long as there are non-suspected processes in the AGREED state, those processes will send NAK(AGREE_FORCED) message in response to ballot messages, and eventually, when no processes fail or become suspected, the root will receive the AGREE_FORCED message.

The distributed consensus algorithm satisfies three properties:

1) (Validity) A process will only commit to a ballot if every non-suspect process accepts it.
2) (Uniform agreement) No two processes will commit to different ballots.
3) (Termination) All non-suspected processes will eventually commit.

We first prove several lemmas then prove that the distributed consensus algorithm satisfies these properties.

Lemma 5: If a root enters Phase 2, then all non-suspect processes have accepted the ballot.

Proof: By inspecting the algorithm there are four ways to start Phase 2: (1) from Phase 1 if a NAK(AGREE_FORCED) is received (line 10), (2) from Phase 1 if the root received an ACK(ACCEPT) from every child (line 15), (3) from Phase 2 if a NAK is received (line 21) and (4) if a new root is appointed and it has received an AGREE message (line 54).

We first prove that if a root enters Phase 2 by cases 1, 3 or 4, then a root must have previously entered Phase 2.

Proof by contradiction: Assume the root enters Phase 2 by case 1, 3 or 4, but no root has previously entered Phase 2.

Case 1: The root has received a NAK(AGREE_FORCED). A process will only send this if it has received an AGREE message, and AGREE messages are only sent in Phase 2, so a root must have been in Phase 2. Contradiction.

Case 3: In this case the root is already in Phase 2. Contradiction.

Case 4: In this case the new root is in the AGREED state indicating it has received an AGREE message, and AGREE messages are only sent in Phase 2, so a root must have been in Phase 2. Contradiction.

We know that if a root is in Phase 2, a root must have at some point entered Phase 2 by case 2, so we only need to prove the statement for case 2. In case 2, the root will enter Phase 2 if the broadcast algorithm returns an ACK(ACCEPT) after broadcasting a ballot. The root will only get an ACK(ACCEPT) from the broadcast algorithm if every process has received the ballot and has agreed to it (by the correctness property of the broadcast algorithm). ■

Lemma 6: If a root enters Phase 3, all processes have received an AGREE message.

Proof: By inspecting the algorithm, there are three ways to enter Phase 3: (1) from Phase 2 if the root receives an ACK

from every child (line 22), (2) from Phase 3 if a NAK has been received (line 28) and (3) if a new root is appointed and it has received a COMMIT message (line 52).

We first prove that if a root enters Phase 3 by cases 2 or 3, then a root must have previously entered Phase 3.

Proof by contradiction: Assume the root enters Phase 3 by case 2 or 3, but no root has previously entered Phase 3.

Case 2: The root is already in Phase 3. Contradiction.

Case 3: If the new root has already received a COMMIT message, a root must have already entered Phase 3, since COMMIT messages are only sent in Phase 3. Contradiction.

We know that if a root is in Phase 3, a root must have at some point entered Phase 3 by case 1, so we only need to prove the statement for case 1. If the root entered Phase 3 by case 1, then the broadcast algorithm must have returned an ACK after broadcasting AGREE messages, therefore every process must have received the AGREE message (by the correctness property of the broadcast algorithm). ■

Lemma 7: If a ballot is acceptable to all processes, and no process fails or becomes suspect during Phase 1, the root will complete Phase 1 and enter Phase 2.

Proof: By the termination property of the broadcast algorithm, the broadcast of the ballot will terminate. If another non-suspect process has received an AGREE message, the root will receive a NAK(AGREE_FORCED) message and enter Phase 2.

If no non-suspect process has received an AGREE message, the broadcast will return an ACK. Since the ballot is acceptable to all processes, an ACCEPT will be piggybacked on the ACK and the root will enter Phase 2. ■

Lemma 8: If no process fails or becomes suspect during Phase 2, the root will complete Phase 2 and enter Phase 3.

Proof: The broadcast of the AGREE message will terminate, and since no process fails or becomes suspect, the broadcast will return an ACK and the root will enter Phase 3. ■

Lemma 9: If no process fails or becomes suspect during Phase 3, all processes will commit.

Proof: The broadcast will terminate and the root will receive an ACK from every child, indicating that every process received the COMMIT message. Since every process received a COMMIT message, every process will have committed. ■

We now prove that the distributed consensus algorithm satisfies the validity, uniform agreement and termination properties.

Theorem 4: (Validity) A process will only commit to a ballot if every non-suspect process accepts it.

Proof: A process will only commit if it receives a COMMIT message, or if it is the root and it enters Phase 3. COMMIT messages are only sent from Phase 3, so a process will only receive a COMMIT if the root has entered Phase 3. We know from Lemma 6 that if the root has entered Phase 3, all processes have received an AGREE message. The root will only enter Phase 3 if it has entered Phase 2. If a root enters Phase 2, then all non-suspect processes have accepted the ballot. Therefore, if a process commits, every non-suspect process has accepted the ballot. ■

Theorem 5: (Uniform agreement) No two processes will commit to different ballots.

Proof: Once a root enters Phase 2, it will never go back to Phase 1, so a root process will never send an AGREE message for more than one ballot.

From Lemma 6, a process will not enter Phase 3 unless every process has received an AGREE message. A process will only commit if it has received a COMMIT message, and COMMIT messages are only sent in Phase 3. So if a process commits to a ballot it must have received an AGREE message.

It follows then that in order for two processes to commit to different ballots, there must be two root processes. This may only happen if the new root process incorrectly suspects the old root process and appoints itself as root while the old root is alive and not suspected by other processes.

If the old root does not suspect the new root and if the old root is not in the COMMITTED state, then it must still need to broadcast the AGREE message. If that broadcast were to return an ACK, then all processes must have received the AGREE message (by the correctness property of the broadcast algorithm). However, because the new root suspects the old root, it will not receive any messages from the old root. This means that the old root will not successfully complete the broadcast algorithm and so cannot enter Phase 3 and enter the COMMITTED state or send COMMIT messages.

If the old root does suspect the new root, then if the broadcast of an AGREE message succeeds at one of the roots, we know that all of the non-suspected processes have the same ballot. If the other root tries to broadcast an AGREE message with a different ballot, it will receive a NAK (line 40) and will not be able to complete the broadcast, and so will not be able to enter Phase 3 to commit or send COMMIT messages. However if the ballots from both roots are the same, the broadcasts of AGREE messages from both roots can succeed, but because both ballots are the same, all processes will commit to the same ballot.

Therefore no two processes will commit to different ballots. ■

Theorem 6: (Termination) All non-suspected processes will eventually commit.

Proof: From Lemmas 7, 8 and 9, we see that as long as processes do not fail or become suspect during a phase, the phase will complete. Since we have assumed that failures (or processes becoming suspect) will cease long enough to allow the algorithm to complete, then regardless of which phase the root is in when the failures cease, all processes will commit. ■

IV. Implementing MPI_Comm_validate

In this section, we describe how the MPI_Comm_validate function can be implemented using the distributed consensus algorithm.

In an implementation of MPI_Comm_validate, the root sends its set of suspected processes from the communicator as the ballot. A process will accept the ballot if it does not suspect any additional processes, otherwise it rejects the

ballot. If the ballot is rejected, the root updates its suspect set and tries again. Eventually the ballot will be acceptable to all processes. We can improve the convergence time if a process were to include the failed processes missing from the ballot in the ACK(REJECT) message. Once the process reaches the COMMITTED state, the process can return from the MPI_Comm_validate call, however, it must periodically check (e.g., in the progress function of the MPI implementation) for the failure of the root. If the root becomes suspect, the process may need to participate in another broadcast of the COMMIT message.

Loose semantics of MPI_Comm_validate can be implemented by eliminating Phase 3 and committing as soon as the process reaches the AGREED state. At this point every process knows that every other process has agreed to the ballot. If the root process does not become suspect before the broadcast of the AGREE message succeeds, then all processes will commit to the same ballot. If, however, the root becomes suspect before the broadcast succeeds and all processes that have received the AGREE message and have committed also become suspect, then the remaining processes may commit to a different ballot than the processes that committed earlier. However, all non-suspect processes will have committed to the same ballot.

We have implemented MPI_Comm_validate as an MPI program rather, than modifying an MPI implementation. This allowed us to evaluate the algorithm at a large scale on a Blue Gene/P. We expect the performance of the operation implemented this way to be an upper bound on the performance of the operation if it were integrated into an MPI implementation.

V. Performance Evaluation

In this section we first analyze the time complexity of the distributed consensus algorithm, then describe our experimental evaluation of the MPI_Comm_validate operation at large scale.

A. Algorithm Analysis

The algorithm has three phases, each consisting of a broadcast and a reduction operation. The complexity of the broadcast and reduction is a function of the number of live processes and depends on the shape of the broadcast tree. In Section III-A, we mentioned that if the compute_children function always chose a child closest to the median from the descendant set, the algorithm would generate a binomial tree.

If we implement the algorithm this way, then every process selects its first child such that it assigns half of its descendants to the first child. That would create a tree with a depth of $\lceil \lg n \rceil$. Since the algorithm performs six broadcasts and reductions on the tree, the algorithm requires $\mathcal{O}(\log n)$ steps to complete in the failure-free case. The implementation evaluated below used this method to construct the broadcast trees.

If we consider failures during the algorithm and the failures occur frequently enough, then theoretically the algorithm might never complete. However in practice we expect that periodically the failures will cease long enough to allow the algorithm to complete at least a phase. That is sufficient to allow the algorithm to make progress.

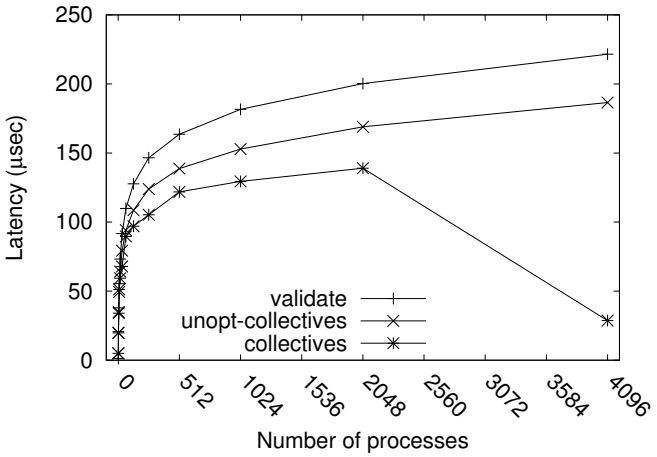

Fig. 1. Comparison of the validate operation with collectives operations performing a similar communication pattern, using Blue Gene/P optimized collectives and unoptimized collectives.

B. Experimental Evaluation

We evaluated our implementation at Argonne National Laboratory on *Surveyor*, a Blue Gene/P with 1,024 quad-core nodes. Figure 1 shows the results of the evaluation. As expected, the operation scales logarithmically. For comparison, we evaluated the time taken to perform a communication pattern similar to that of the validate operation using broadcast and reduction operations. The figure shows the results with optimized collectives using the Blue Gene/P collective tree network and with unoptimized collectives using the same torus network that the validate operation uses. At full scale, the validate implementation took 222 μs to perform the operation, which is 1.19 times slower than performing a similar communication pattern with unoptimized collectives. We expect the performance of the validate algorithm to improve when the operation is integrated into the MPI implementation by making the algorithm more responsive to incoming messages.

We also evaluated the performance of the operation with loose semantics. Figure 2 shows the comparison. Using loose semantics, the operation is performed 94 μs faster at full scale than the strict implementation (a speedup of 1.74). Depending on the requirements of the application and the frequency at which the application calls validate, using the loose implementation can provide performance improvement to the application.

We evaluated the performance of the MPI_Comm_validate with failed processes. We started with 4,096 processes then randomly chose processes to fail. Figure 3 shows the performance of MPI_Comm_validate with strict and loose semantics while the number of failed processes was varied between zero and 4,095. The graph shows a jump in latency between zero and one process failure. This is due to the fact

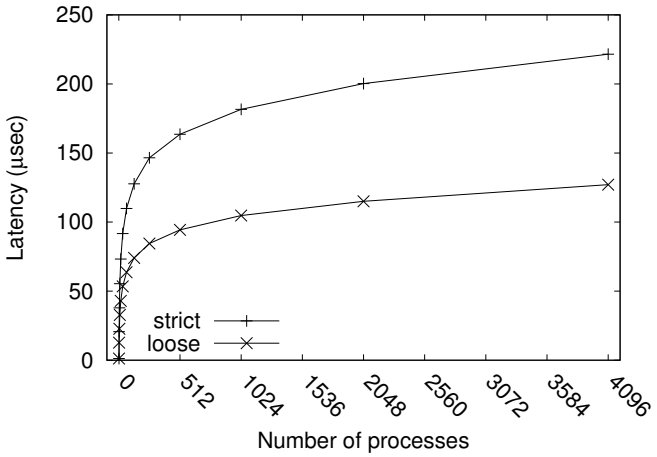

Fig. 2. Comparison of validate using strict and loose semantics.

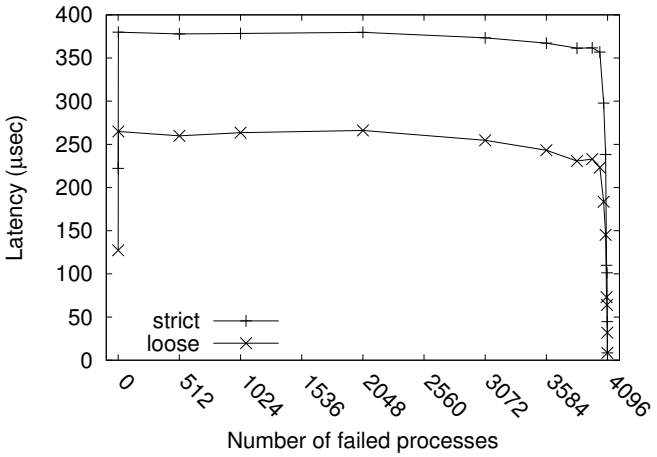

Fig. 3. Performance of validate with failed processes.

that in the failure free case, the list of failed processes is not sent. However, in the case where failed processes have been detected, the bit vector representing the list of failed processes is sent as a separate message in Phases 2 and 3 of the algorithm. Each non-root process then needs to compare this list to its local list of failed processes. All of this adds overhead to the operation. A possible optimization to be investigated would be to use a different, more compact, representation of the list, e.g., an explicit list of failed processes rather than a bit vector, when the number of failed processes is below a certain threshold.

Notice that as the number of failed processes increases from 1, the latency stays relatively constant until around 3,600 failed processes. This is due how the broadcast algorithm generates the tree. With failed processes, the shape of the tree remains close to that of a binomial tree with no failed processes and so has similar depth. However after around 3,600 failed processes, the depth of the tree quickly decreases, which decreases the latency of the algorithm.

VI. Related Work

Chandra-Toueg [5] and Paxos [6] are the classical methods for achieving distributed consensus. These algorithms have scalability issues in that the coordinator process sends and receives messages individually from every process. Work has been done to improve scalability in [7] and [8]; however, these solutions are targeted for database systems that might have only tens or hundreds of committing processes in a large-scale system and so are not appropriate for exascale systems. The algorithm presented in this paper uses a fault tolerant broadcast tree to distribute and collect messages, making the algorithm highly scalable. The Paxos algorithm is tolerant to network partitioning, which is a failure mode not considered in this paper.

In [9], Weikum and Vossen describe a "transactional tree two-phase commit algorithm" where commit and acknowledgment messages are forwarded to committing processes over a tree. This is a similar approach to our algorithm. However Weikum and Vossen do not describe how the tree is to be constructed dynamically nor how to handle process failures while the messages are being propagated over the tree. These issues are explicitly addressed in this paper.

In [10], Fischer et al. proved that distributed consensus in an asynchronous model with one faulty process is impossible in a finite number of steps. Our algorithm does not guarantee consensus in a finite number of steps; rather, it will reach consensus with a probability of 1.

Hursey et al. described an implementation of the loose semantics of the MPI_Comm_validate operation in [11]. The implementation is based on the two-phase commit algorithm and requires a termination-detection algorithm to avoid blocking when the coordinator fails. Their algorithm uses a static tree structure that is preserved between invocations of the MPI_Comm_validate operation. When a process fails, children of the failed process search for a live ancestor and reconnect to it. After the MPI_Comm_validate operation completes, the tree is rebalanced to compensate for any failed processes. If the coordinator fails before a child sends its vote, that child process decides to abort. If, however, the coordinator fails after a child has sent its vote but before receiving a decision, it queries its siblings (i.e., every other child of the failed coordinator) to determine if any has reached a decision. If so, the child makes the same decision and broadcasts that decision down its subtree. Their algorithm is also log-scaling, but does not implement strict semantics.

VII. Conclusion

This paper presented a scalable distributed consensus algorithm used to implement the MPI_Comm_validate operation proposed by the MPI 3 fault-tolerance working group. This paper also presented proofs that the algorithm satisfies the validity, uniform agreement and termination properties. The algorithm was evaluated on a 4,096-core Blue Gene/P machine and was shown to be extremely scalable. The implementation was able to perform a full-scale validate operation in 222 μs and scaled logarithmically.

Using loose semantics, the implementation showed a speed-up of 1.74 compared to strict semantics. Depending on the requirements of the application, loose semantics may be appropriate and can reduce the impact of adding fault-tolerance features to an application.

We intend to implement the MPI_Comm_validate operation in MPICH2. We expect that this implementation will improve the responsiveness of the algorithm and hence improve its performance. Furthermore, we intend to use a similar algorithm to implement other operations requiring distributed consensus, such as the communicator creation routines.

ACKNOWLEDGMENTS

We thank Joshua Hursey for his suggestions and ideas on designing and implementing MPI_Comm_validate. We also thank the members of the MPI Forum and the fault-tolerance working group, especially Torsten Hoefler for his helpful comments on the paper.

REFERENCES

[1] J. Anfinson and F. T. Luk, "A linear algebraic model of algorithm-based fault tolerance," *IEEE Transactions on Computing*, vol. 37, pp. 1599–1604, 1988.

[2] Z. Chen and J. Dongarra, "Algorithm-based fault tolerance for fail-stop failures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 12, December 2008.

[3] ——, "Highly scalable self-healing algorithms for high performance scientific computing," *IEEE Transactions on Computers*, July 2009.

[4] Fault Tolerance Working Group, "Run-though stabilization proposal." [Online]. Available: http://svn.mpi-forum.org/trac/mpi-forum-web/wiki/ft/run_through_stabilization_2

[5] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, pp. 225–267, March 1996.

[6] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, pp. 133–169, May 1998. [Online]. Available: http://doi.acm.org/10.1145/279227.279229

[7] S. Ranganathan, A. D. George, R. W. Todd, and M. C. Chidester, "Gossip-style failure detection and distributed consensus for scalable heterogeneous clusters," *Cluster Computing*, vol. 4, pp. 197–209, July 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1011494323443

[8] P. Jurczyk and L. Xiong, "Adapting commit protocols for large-scale and dynamic distributed applications," in *Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems:*, ser. OTM '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 465–474. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-88871-0_33

[9] G. Weikum and G. Vossen, *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., 2001, ch. 19.3, pp. 744–748.

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, pp. 374–382, April 1985. [Online]. Available: http://doi.acm.org/10.1145/3149.214121

[11] J. Hursey, T. Naughton, G. Vallee, and R. Graham, "A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2011, vol. 6960, pp. 255–263. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24449-0_29